A Jupyter Kernel for Prolog

Anne Brecklinghaus[1], Philipp Körner[1]

Abstract: Benefits of literate programming are well-known: By combining source code and documentation, concepts can be introduced in a more comprehensible way to the reader. Motivated by rendering our Prolog applications and teaching examples more readable, we implemented a Jupyter kernel for SICStus Prolog. In this paper, we will give a feature overview of this kernel, discuss particularities caused by Prolog's execution mechanism and report on our experiences porting the kernel to SWI-Prolog.

Keywords: Literate Programming; Prolog; Jupyter Notebook; Integration

1 Introduction and Motivation

In 1984, Donald Knuth, the creator of TEX, developed the notion of *literate programming* [Kn84]. The idea is that programs and their documentation should be seen as *works of literature*. A good explanation should facilitate understanding the program and therefore result in less time spent for debugging. For this, the programming language WEB was developed with the aim of combining code in Pascal and documentation in TEX in one file.

Jupyter notebook documents are a more recent implementation of this idea [PG07, Pra]. They are convenient for executing code interactively as well as for documentation, which is why they have become popular in scientific contexts. Notebooks can be viewed, edited, and converted into other formats with **Jupyter** applications such as **Jupyter Notebook** and **JupyterLab**. These frontends communicate with a kernel that determines the programming language of the notebook and is responsible for code execution.

Our group in Düsseldorf uses a combination of SWI-Prolog [Wi12] and SICStus Prolog [CM12] for teaching and also maintains the formal methods tool ProB [LB08] (a model checker and contraint solver that makes heavy use of SICStus-specific features, in particular the strong CLP(FD) solver; the core functionality was ported to SWI-Prolog [GL22]). Many student theses revolve around the ProB tool, yet documentation is sparse and distributed between the source code, a wiki and a handbook. While for SWI-Prolog, the excellent **SWISH** tool [Bo16, Wi19] exists, we could only locate a very old Jupyter kernel for SICStus 3. Thus, in order to improve our teaching material for self-studying (we already made good experiences with **Jupyter** notebooks in courses on theoretical computer

[1] Heinrich-Heine-Universität Düsseldorf, Institut für Informatik, Universitätsstraße 1, 40225 Düsseldorf, Germany
{anne.brecklinghaus,p.koerner}@hhu.de

cc by sa

science and safety critical systems using a custom kernel for ProB itself [GL22]) and the documentation of ProB, we implemented a **Jupyter** kernel for SICStus Prolog. Later, we modified it to support SWI-Prolog and be highly customisable so that it can be extended for further Prolog implementations. This kernel and notebooks describing the kernel's features for SICStus and SWI-Prolog by providing examples is available at:

`https://github.com/anbre/prolog-jupyter-kernel`

In the following, we report on our experiences during this implementation. First, we introduce the reader to **Jupyter** and existing applications in Sect. 2. Afterwards, in Sect. 3, we briefly describe the architecture of the tool. In Sect. 4, we give an overview of the features of our notebook application. Further, we report on our experiences on porting the application to SWI-Prolog in Sect. 5, before giving our conclusions in Sect. 6.

2 Background & Related Work

In this section, we will first provide an introduction to **Jupyter** notebooks and their eco-system. Afterwards, we discuss related applications and existing kernels for Prolog.

2.1 Background

Jupyter notebooks[2] originate from the **IPython** project [PG07] with the goal of enabling interactive Python development. A **Jupyter** notebook consists of *cells* which can contain either executable code (and its output) or accompanying rich text which is not meant for execution. Each code cell is separate from the others in that its execution only influences its own output (aside from global program state modifications). The successor **Project Jupyter** [Pra] was developed in order to support notebooks for other programming languages.

Each **Jupyter** notebook is associated with a so-called *kernel* which determines the programming language in which code can be executed. As can be seen in Fig. 1, when a user interacts with a **Jupyter** frontend, it sends a corresponding request to the connected kernel. After handling the request, the kernel sends a reply to the frontend which needs to be handled and displayed to the user. While the default IPython kernel is the most well-known, there are other popular kernels such as **IRkernel** for the R language [KA] and **IJulia** for Julia [La], and many community-maintained kernels for various languages [Prb].

Jupyter Notebook [Jub] was the first web application with which these documents could be created and viewed. Later, the highly customisable **JupyterLab** [Prc] was released, which is planned to replace **Jupyter Notebook** eventually. In addition to notebook applications, there

[2] Not to be confused with the frontend application "**Jupyter Notebook**".

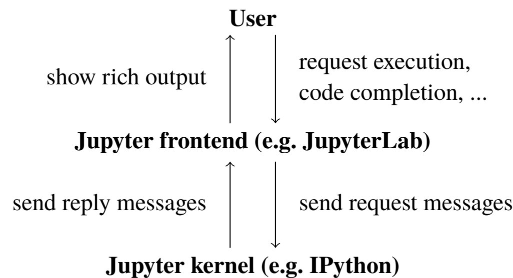

Fig. 1: Diagram showing how user interaction with a Jupyter frontend is handled

are console frontends like **Jupyter Console** [Tha] and **QtConsole** [Thb]; they behave similar to an interactive Prolog session[3], but can inherit Jupyter features like code completion and provide readline-wrapping (e.g., making the arrow keys usable on the terminal and adding a searchable history of commands) by default. Furthermore, the Jupyter applications **Voilà** [Thc], **nbviewer** [Prd], and **nbconvert** [Jua] facilitate the distribution of notebook documents, e.g., by generating presentations, LaTeX, or static HTML documents. All **Jupyter** applications can be used with any programming language for which a kernel exists.

Indeed, the **Jupyter** notebooks only offer a single cell type for code, which aligns with the fundamentals of a typical imperative programming language. This often is a restriction that requires workarounds in several aspects of our kernel: Prolog, as a declarative language, differs significantly in its control flow (e.g., backtracking, coroutining, etc.). Nonetheless, the basic workflow — typing in queries and getting some data as answer — can still be implemented on top of the notebook infrastructure. For a typical notebook style, however, one should also have the possibility to extend and modify the program itself. This requires to distinguish between program code and queries. In order to keep compatibility with all frontend applications, any interaction with Prolog still has to occur in the single code cell type, rendering notebook extensions — such as different cell types for program code and queries, buttons to cycle through several solutions, etc. — infeasible.

2.2 SWISH and Community Prolog Kernels

SWISH [Wi] is a notebook application developed for SWI-Prolog and it is meant to facilitate sharing Prolog code. When saving a file a user can decide if it is public, in which case other users can search for it. Cells can contain a `Program`, `Query`, `Markdown`, or `HTML` text. Code defined in a `Program` cell can be configured to be either callable from queries below the cell only or from all queries in the notebook. `Query` cells can be run and the number of desired solutions can be defined. In addition to that, queries can also be executed in a separate Prolog REPL. If a query succeeded with a choice point, additional solutions can be

[3] We will refer to interactive sessions as REPL (read-eval-print-loop).

requested. The results can be displayed as a table as well as be downloaded as a CSV file. The whole notebook can be printed and thereby converted to a PDF file.

When running a query in the aforementioned REPL, information about some elements can be accessed by hovering over them. For example, for own predicates, the line of the first defined clause is shown and a description can be seen for SWI-Prolog predicates. For these queries, some interactive debugging features are available.

SWISH supports most functionality of Prolog (with some limitations due to security concerns) including producing output. While it additionally implements some more helpful features, such as real-time collaboration, it also lacks some functionality the **Jupyter** applications provide (e.g., exporting slides). Moreover, it was implemented solely for SWI-Prolog and — opposed to **Jupyter** — it is not meant for being extended for any other programming language.

Furthermore, multiple **community-written Prolog kernels** are available, many of which, are written for outdated Prolog versions and unmaintained. As often basic functionality is missing and they were not implemented with extensibility in mind, we implemented our own kernel from scratch. Nonetheless, in the following, we will present a selection.

The **Calysto Prolog** kernel [Ca] executes code based on a Prolog interpreter written in Python [MO]. Each term ending with a `?` is interpreted as a query and other terms are used to define new Prolog facts and rules. The kernel can be used to define and query simple facts and even compute alternative solutions with a magic command. Still, basic functionality such as querying rules and producing output does not seem to work and no helpful error messages are printed.

Two **Jupyter** kernels for SWI-Prolog stand out: the **SWI-Prolog-Kernel** [Me] which in turn inspired **jswipl** [Co]. However, neither kernel provides more advanced features such as producing output or using DCGs.

For the **SWI-Prolog-Kernel**, no proper installation instructions exist. It seems to work by writing Prolog code to a file and executing the file with SWI-Prolog in a subshell. This means that all cells are independent of each other, and defining clauses in one cell and querying them in another is impossible.

The other SWI-Prolog kernel **jswipl** uses the SWI-Prolog and Python interface **PySwip** [Tc] for code execution. Contrary to **PySwip**, each fact or rule which is added exists as long as the kernel is running. Therefore, cells containing facts and rules should not be executed more than once and programs cannot be altered. A query needs to start with `?-` and if there is more than one solution, by default, up to 10 answers are printed (though this limit can be adjusted with special syntax). The **jswipl** kernel does not claim to be tested properly and **PySwip** does not claim to be complete.

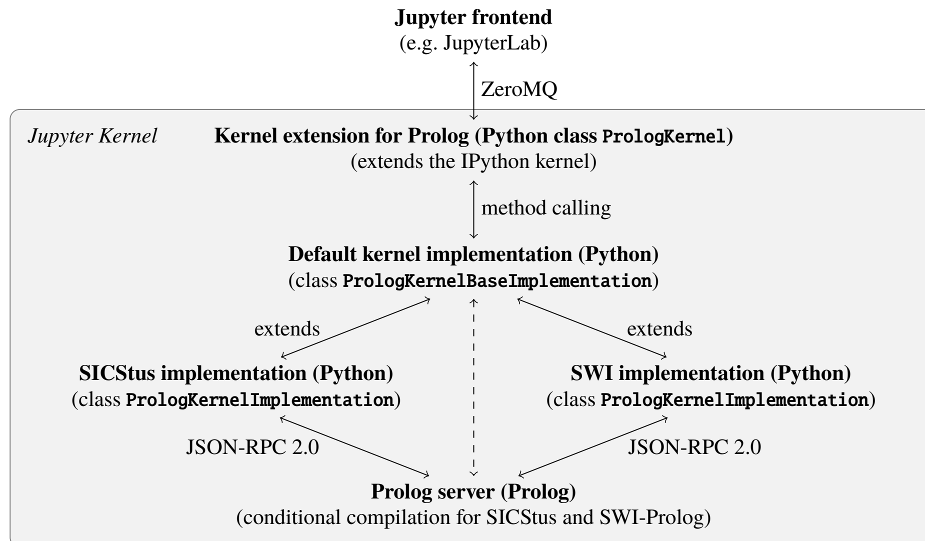

Fig. 2: Diagram showing the architectural components and their communication methods

3 Architecture

Fig. 2 provides an overview over the basic architecture of our implementation: The Jupyter kernel is split in three: the first part is written in Python and interacts with an arbitrary Jupyter frontend via the ZeroMQ protocol [Hi13]. We extended the IPython kernel so that it handles a Prolog backend and inherits the communication with the frontend. Note that this extension does not interpret Prolog itself.

Instead, the third part, an instance of (SWI- or SICStus) Prolog is started. That Prolog server is based on a recent addition to SICStus Prolog, a library that handles JSON remote procedure calls (RPCs). The kernel starts a Prolog subprocess and communicates with it according to the JSON-RPC 2.0 protocol. For any code execution request the kernel receives from the frontend, it sends a request message to the Prolog server containing the code

Prolog terms are read from the code with `read_term/3` or `read_term_from_atom/3` depending on the Prolog implementation. In general, all terms are looped over and after handling one, the loop is continued with the next one (the main idea is depicted in Fig. 3). However, if a query execution is

```
server_loop :-
  next_term(Term),
  (Term == call(X) -> call(X), server_loop
  ; (Term == retry -> fail
    ; (Term == cut -> ! ; ...))),
  server_loop.
```

Fig. 3: Rough overview of the server loop

successful, instead of continuing the current loop, a new recursive loop is started. In that case, the current goal is seen as the active one which can be retried. This is possible because the predicate which actually calls the goal leaves a choicepoint. When a **retry** request is received, the current loop iteration fails, causing the active goal to backtrack. If a **cut** is encountered instead, choicepoints of the active query are cut off and the current loop terminates.

By using the option **variable_names(Variables)** when reading the Prolog terms, a list of **Name=Var** pairs is obtained. After all terms are processed, the server sends a response. For instance, clause definitions are asserted and queries are executed with **call/1**. In the latter case, the list of variables which might have been bound is included in the response. Depending on the type of the server response (e.g., solutions are found or errors raised), the kernel sends the response to the frontend which displays it to the user.

The tool can be extended in order to support additional Prolog interpreters: When the kernel is started, it loads a configuration file which contains the path to interpreter-specific Python class files. Such classes are responsible for starting and communicating with their corresponding Prolog server. For every request the kernel receives, a method of the implementation class is called. Therefore, to configure a different interpreter (which needs to extend a base class with the default implementation), the class can easily be overridden (e.g., to support another Prolog interpreter). Additionally, the kernel can be configured to start a different implementation of the Prolog server[4].

4 Feature Overview

In this section, we will present the features of our Jupyter kernel. First, we consider particularities of Prolog that stem from using SLD-resolution as its execution mechanism. Second, we present non-standard features that are available in almost every Prolog implementation. Last, we show features that are driven by the Jupyter integration and are provided mostly for user convenience.

Where appropriate, we will briefly touch on implementation details if they are caused by our choice of architecture. Note that most special predicates in the `jupyter` module need to be executed as the only predicate in a cell. Some details may differ based on the Prolog interpreter that is used and cannot be listed in detail here (but can be found in the notebooks in the GitHub repository).

[4] The GitHub repository provides an explanation of all available configuration options as well as an example configuration file.

4.1 Prolog Particularities

As discussed in Sect. 2.1, a typical Jupyter application is limited to a single code cell type only. However, each term in a code cell might be either a clause definition or a query (that might be in the form of a directive) which shall be evaluated (Fig. 4). While terms like directives and clauses with bodies can easily be distinguished from queries, it is more difficult for clauses without bodies. One option for differentiation would have been to expect all queries to be somehow marked. However, this would likely cause frustration for users forgetting about it because they are used to a Prolog REPL where no such marking is required.

Therefore, if a cell contains a single potential query, it is interpreted as such instead of a clause definition. If users want to assert a single fact, they can still achieve it by writing `foo :- true.`. Additionally, terms starting with `?-` and directives starting with `:-` are evaluated as queries (which corresponds to valid Prolog syntax) even in a cell containing further terms. If instead, a potential query without prefix or body is encountered with multiple other terms, it is handled as a clause definition. Further, terms with bodies are always seen as clause definitions. In each case, as described below, the user can infer from the output how a term was interpreted.

Predicate (Re-)Definition: The possibility of defining programs on the fly is a major advantage of using Prolog with a Jupyter notebook over a Prolog REPL. Except from `plunit` test definitions, all defined clauses are added as dynamic facts to the data base[5]. In order to let the user know that a predicate was (re-)defined, a message is output with the corresponding predicate specification.

```
[1]: app([], Res, Res).
     app([Head|Tail], List, [Head|Res]) :-
       app(Tail, List, Res).

     % Asserting clauses for user:app/3

[2]: app([1,2], [3,4], R).

     R = [1,2,3,4]
```

Fig. 4: Predicate definition and query

Further, interactive programming involves writing, testing and re-writing clauses rather than only adding new clauses to the fact database. Therefore, by default, when clauses are defined for a dynamic predicate for which there are existing ones, these are retracted first (in this case, the user is informed). Thus, all clauses of a predicate need to be defined in one cell. However, when a predicate is declared **`discontiguous`**, new clauses are added without any preceding retractions.

Query Execution: If a query succeeds and binds any variables, the bindings are usually shown in the output of the cell like they would be displayed in a console. Analogously, if there are no bindings or the query fails, the corresponding output for success or failure is displayed (e.g., **`yes`** or **`no`**)[6]. During execution, any output is redirected to a file so that it can

[5] Modules loaded from disk are not added as dynamic facts.

[6] In order to mimic the Prolog behavior, no such result is displayed for directives.

be read in again and displayed to the user preceding the query result. Additionally, if an exception is caught, the corresponding message is computed and output in red.

Handling Multiple Solutions: When a query succeeds with a choice point, one can request further solutions via backtracking. The Jupyter kernel mimics this by providing the predicate `jupyter:retry/0`: Whenever a query is executed, it is seen as the active query as long as there might be further solutions for it. Then, the `retry` predicate may be called from within the same cell as the query or, alternatively, in a new cell. Backtracking is triggered to compute the next solution by causing a loop failure (see Sect. 3). Additionally, the predicate `jupyter:cut/0` is exposed to cut off choice points of the active execution and to set an older query which might have open choice points as active. The stack of these queries can be inspected with `jupyter:print_stack/0`.

Term Expansion: With many Prolog implementations, definite clause grammars (DCGs) can be defined by writing so-called grammar rules of the form `Head --> Body.`. When a Prolog file containing such rules is compiled, they are automatically translated into Prolog clauses by passing them to a term expansion predicate. The Prolog Jupyter kernel handles grammar rules similarly. For every rule which is encountered, term expansion is manually applied in order to compute the resulting clause. This clause can then be handled in the same way as any other clause definition.

4.2 De-Facto Standard Prolog Features

Loading Source Files and Libraries: Loading source files and libraries works in the same way as on a Prolog REPL. Predicates are always re-defined when encountered (as user interaction is not possible during the load). Note, however, that loading a library *and* using operators from that library does not work in a single query, as it needs to be read by a built-in read predicate, which throws a syntax error when encountering undefined operators.

Test Runner: `library(plunit)` can be used to define and run automated tests. Tests can either be defined in a file which is loaded or in a cell. Note that `test/1` or `test/2` are the only predicates which cannot simply be asserted as dynamic clauses. Instead, whenever such a clause is encountered after a `begin_tests` directive, it is written to a file which later is loaded. Therefore, in order to be recognised as such, any test definition needs to be preceded by a `begin_tests` directive. Additionally, if there is an optional `end_tests` directive, it needs to follow the test clauses.

Benchmarking Capabilities: Whenever a query is executed, its runtime is stored in a data base. It can be output with `jupyter:print_query_time/0`, which prints the latest previous query and its runtime in milliseconds measured with `statistics(walltime, Value)`.

Debugging: In a Jupyter notebook, debugging cannot be performed interactively as user input is not supported during the execution of a cell. Thus, switching on trace mode with `trace/0` would cause the server to stop at an invocation (and a restart is required).

However, the call stack can be accessed by printing debugging messages with breakpoints. As this mechanism might be difficult for students, we implemented the predicate **`jupyter:trace(Goal)`**. It switches on trace mode, calls the goal **`Goal`** and switches debug mode off. By default, all ports are unleashed and included in the output (Fig. 5), so that no user interaction is requested on breakpoint activation.

```
[3]: jupyter:trace(app([1,2], [3], R)).

          1      1 Call: app([1,2],[3],_188345)
          2      2 Call: app([2],[3],_192175)
          3      3 Call: app([],[3],_192623)
          3      3 Exit: app([],[3],[3])
          2      2 Exit: app([2],[3],[2,3])
          1      1 Exit: app([1,2],[3],[1,2,3])
R = [1,2,3]
```

Fig. 5: A trace for append

4.3 Jupyter Convenience Features

Introspection: In JupyterLab as well as Jupyter Notebook, *code completion* for the token at the current cursor position can be requested by pressing the Tab key. If there is a single possible match, the code preceding the cursor is replaced directly. Otherwise, a list of options is shown from which the user can choose one (Fig. 6). Completion can be used for predicates which are built-in or exported by a loaded module[7]. Requesting completion for a module name is especially useful for retrieving all special **`jupyter`** predicates. After loading another module, completion data can be updated with **`jupyter:update_completion_data/0`** so that completion works for predicates from that module as well.

By pressing Shift+Tab, inspection for the token preceding the current cursor position can be requested. For SWI-Prolog, documentation for the token is retrieved with **`help/1`** and shown right away. However, this is not possible for SICStus Prolog. Instead, for all predicates which the Predicate Index page [Pre] lists, a link to the documentation of the corresponding predicate is shown if the predicate's name contains the current token. The data shown about the predicate is the same as shown on the web page, which includes the predicate's name and arity. For JupyterLab, clickable hyperlinks can be displayed; since this is not possible for Jupyter Notebook, the links are only given to be copied. However, module **`jupyter`** defines documentation for its exported predicates. This can be shown right away for inspection and also be output with **`jupyter:help/0`**.

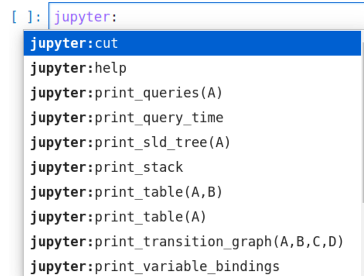

Fig. 6: Code completion

Accessing Previous Results: SWI-Prolog provides functionality of reusing top-level bindings. Roughly, when a top-level goal succeeds, its bindings are asserted in a database (i.e., succeeding calls will update the binding). Before calling a top-level query, it is expanded

[7] These are loaded when the kernel is started.

to replace any **$Var** term with the stored binding of variable **Var**. This way, the latest bindings from previous queries can be accessed. The Jupyter kernel also provides this functionality for SICStus Prolog. Additionally, there is the predicate **jupyter:print_variable_bindings/0** that outputs all stored variable bindings.

Further, all executed queries are stored internally: When writing a new predicate, a user might test its subgoals gradually in different cells, potentially using **$Var** terms to access previous values. Once all the parts are written, the predicate **jupyter:print_queries(Ids)** can be called to access previous queries from cells with IDs in the list Ids. They are printed in a way that they can easily be copied to a cell and executed right away (Fig. 7) or expanded with a head to define a predicate. If a query contains a **$Var** term and one of the previously printed queries contains the variable **Var**, the term is replaced by the variable name.

```
[1]: member(Member, [1,2,3]).
Member = 1

[2]: Square is $Member * $Member.
Square = 1,
Member = 1

[3]: jupyter:print_queries([1,2]).
member(Member,[1,2,3]),
Square is Member*Member.
yes
```

Fig. 7: Incrementally building a predicate

Structured Output: Our kernel implementation provides two special predicates to output tables: The predicate **jupyter:print_table(Goal)** can be used to compute all results of the goal **Goal** with **findall/3** and display them in a table. The table contains a column for each variable occurring in it (which were extracted when reading in the term) and a line for each result (Fig. 8). If the goal does not terminate, no data at all can be sent to the client and therefore displayed. In that case, the server needs to be restarted.

In order to fill a table with data that is not obtained by calling **findall/3**, there is the predicate **jupyter:print_table(ValuesLists, VariableNames)**. **ValuesLists** is expected to be a list of lists where each of them corresponds to one line of the table. Therefore, all the lists need to be of the same length. Furthermore, **VariableNames** is used to provide the column headers and needs to be a list of ground terms of the same length as well unless it equals **[]**. In the latter case, the headers contain capital letters starting from **A**.

```
[1]: jupyter:print_table((
         member(Member, [10,20,30]),
         Square is Member*Member)).
```

Member	Square
10	100
20	400
30	900

yes

Fig. 8: Tabular output

Switching Between Prolog Implementations: If implementation-specific data is configured for more than one Prolog implementation, the active Prolog implementation used for code execution can be changed with **jupyter:set_prolog_impl(ImplementationID)**. It needs to be noted that the implementation is changed after all code of the cell has been executed. Therefore, any code following such a query is still executed with the previous active implementation.

The server for the previously used implementation is kept running so that when changing back, the state has not changed. For example, the previous variable bindings still exist. When the Jupyter kernel is interrupted, all running Prolog servers are killed and need to be restarted the next time code is executed. In order to restart a single Prolog process, `jupyter:halt/0` can be used.

5 Porting the Application to SWI-Prolog

After our kernel supported SICStus Prolog, we made adjustments to also support SWI-Prolog (and be extensible for further Prolog implementations). The kernel can be configured to use either interpreter and even switch between them on the fly. However, there was a lot of Prolog code which had to be adjusted. Since similar issues might occur when extending this kernel for another Prolog implementation, our experience might serve as a rough tutorial.

For code execution, the Jupyter kernel can communicate with any Prolog server process over JSON-RPC 2.0. By replacing the Prolog server with another one, the *Python part* of the kernel can easily support a different Prolog implementation. If the replacement of the server does not suffice, most of the Python code can be extended as well[8]. In case of SWI- and SICStus Prolog, the only Python code that differs is for predicate inspection.

The main requirement for the *Prolog server* for a different Prolog implementation is that it can receive requests in the form of JSON-RPC 2.0 messages, handle them and send responses. Most of the code we used for this was compatible with SWI-Prolog[9]. We expect this to be similar for other Prolog implementations, but in rare cases — or when trying to support Prolog derivatives such as Mercury [SHC96] — it might be required to write a new server from scratch.

While implementing the basic code execution does not require major effort, more advanced features such as introspection or other functionality provided by the module `jupyter` may require more significant changes. Some issues we encountered are:

- Some predicates which are built-in for one Prolog implementation need to be loaded from a library for the other one. Furthermore, libraries are named differently and predicates with the same functionality are provided under different names.
- Even though the SICStus `library(plunit)` is based on the one developed for SWI-Prolog and the basic functionality overlaps, there are major discrepancies between them. These include options which can be provided for `begin_tests/2`, `test/2`, and `run_tests/2`. Further, there are different requirements for loaded files defining tests.

[8] The GitHub repository contains a detailed explanation of the configuration and extension options.

[9] Therefore, the Prolog source files contain code for both implementations by using conditional compilation with the directives `if/1`, `else`, and `endif`.

- Reading terms from an atom is more complex in SWI-Prolog when a helpful error message is required (e.g., in case of syntax errors).
- Debugging in general and breakpoint handling differs considerably.
- For SICStus Prolog, a website listing links to documentations of all predicates is used for providing inspection information. Instead, for SWI-Prolog, there is the predicate `help/1` with which information about predicates can be retrieved.

6 Conclusions and Future Work

In this paper, we presented an implementation of a Jupyter kernel for Prolog developed for SICStus Prolog and its extension to accommodate SWI-Prolog. Communication with a Prolog interpreter follows a JSON-RPC protocol. Overall, we think it has great merits to share code examples for teaching or for tutorials on usage of larger libraries or applications.

The execution mechanism of Prolog does not follow the typical statements that Jupyter expects. Especially distinguishing between program definitions and queries is finicky when considering user convenience. The notebook application SWISH offers a clean solution with multiple cell types.

Porting the basic functionality of the kernel to SWI-Prolog was relatively easy, yet more advanced features such as debugging required significant work. Standards for library and introspection would have facilitated this work considerably.

6.1 Future Work

In the future, we want to explore an idea that stemmed from discussions about the portability of Prolog programs and future developments of Prolog itself [Kö22]: The Jupyter kernel can alread by connected with *multiple* Prolog instances and it should be relatively easy to reuse results from one instance for the other. Then, one could combine the individual strengths of several interpreters (e.g., making use of the many SWI-Prolog libraries that handle file formats and then use the efficient CLP(FD) implementation of SICStus Prolog).

Further, one could send commands to all available Prolog interpreters *at once*. Then, the benchmarking infrastructure could be helpful to quickly compare the performance of several Prolog interpreters and identify performance improvements or degradations between multiple versions of the same Prolog implementation. Finally, once a broader range of interpreters is available, it could provide a tool for Prolog implementors and (ISO) standard maintainers to write test suites and locate instances in which Prolog interpreters differ in their behaviour.

Bibliography

[Bo16] Bogaard, Tessel; Wielemaker, Jan; Hollink, Laura; van Ossenbruggen, Jacco: SWISH DataLab: A Web Interface for Data Exploration and Analysis. In (Bosse, Tibor; Bredeweg, Bert, eds): Proceedings BNAIC. volume 765 of Communications in Computer and Information Science. Springer, pp. 181–187, 2016.

[Ca] Calysto: Calysto Prolog. `https://github.com/Calysto/calysto_prolog`. Last access: September 6, 2022.

[CM12] Carlsson, Mats; Mildner, Per: SICStus Prolog – The first 25 years. Theory and Practice of Logic Programming, 12(1-2):35–66, 2012.

[Co] Corbatto, Luca: jswipl. `https://github.com/targodan/jupyter-swi-prolog`. Last access: September 6, 2022.

[GL22] Geleßus, David; Leuschel, Michael: Making ProB Compatible with SWI-Prolog. Theory and Practice of Logic Programming, p. 1–15, 2022.

[Hi13] Hintjens, Pieter: ZeroMQ: Messaging for Many Applications. O'Reilly Media, Inc., 2013.

[Jua] Jupyter Development Team: nbconvert Documentation. `https://nbconvert.readthedocs.io/en/latest/`. Last access: September 6, 2022.

[Jub] Jupyter Team: Jupyter Notebook Documentation. `https://jupyter-notebook.readthedocs.io/en/stable/notebook.html`. September 6, 2022.

[KA] Kluyver, Thomas; A., Philipp: IRkernel. `https://irkernel.github.io/`. Last access: September 6, 2022.

[Kn84] Knuth, Donald E.: Literate Programming. The Computer Journal, 27(2):97–111, January 1984.

[Kö22] Körner, Philipp; Leuschel, Michael; Barbosa, João; Costa, Vítor Santos; Dahl, Verónica; Hermenegildo, Manuel V.; Morales, Jose F.; Wielemaker, Jan; Diaz, Daniel; Abreu, Salvador; Ciatto, Giovanni: Fifty Years of Prolog and Beyond. Theory and Practice of Logic Programming, pp. 1–83, 2022.

[La] Language, The Julia Programming: IJulia. `https://github.com/JuliaLang/IJulia.jl`. Last access: September 6, 2022.

[LB08] Leuschel, Michael; Butler, Michael: ProB: an automated analysis toolset for the B method. Software Tools for Technology Transfer, 10(2):185–203, 2008.

[Me] Mensing, Max: SWI-Prolog-Kernel. `https://github.com/madmax2012/SWI-Prolog-Kernel`. Last access: September 6, 2022.

[MO] Meyers, Chris; Obermann, Fred: Prolog in Python - Part 3. `http://openbookproject.net/py4fun/prolog/prolog3.html`. Last access: September 6, 2022.

[PG07] Pérez, Fernando; Granger, Brian E.: IPython: a System for Interactive Scientific Computing. Computing in Science and Engineering, 9(3):21–29, May 2007.

[Pra] Project Jupyter: `https://jupyter.org/`. Last access: September 6, 2022.

[Prb] Project Jupyter: Jupyter kernels. https://github.com/jupyter/jupyter/wiki/Jupyter-kernels. Last access: September 6, 2022.

[Prc] Project Jupyter: JupyterLab Documentation. https://jupyterlab.readthedocs.io/en/stable/. Last access: September 6, 2022.

[Prd] nbviewer. https://nbviewer.org/, Last access: September 6, 2022.

[Pre] Prolog, SICStus: Predicate Index. https://sicstus.sics.se/sicstus/docs/latest/html/sicstus.html/Predicate-Index.html. Last access: September 6, 2022.

[SHC96] Somogyi, Zoltan; Henderson, Fergus; Conway, Thomas C.: The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. Journal of Logic Programming, 29(1-3):17–64, 1996.

[Tc] Tekol, Yüce; contributors: PySwip. https://github.com/yuce/pyswip. Last access: September 6, 2022.

[Tha] The Jupyter Development Team: Jupyter console. https://jupyter-console.readthedocs.io/en/latest/. Last access: September 6, 2022.

[Thb] The Jupyter Development Team: Jupyter QtConsole. https://qtconsole.readthedocs.io/en/stable/index.html. Last access: September 6, 2022.

[Thc] The Voilà Development Team: Voilà Documentation. https://voila.readthedocs.io/en/stable/. Last access: September 6, 2022.

[Wi] Wielemaker, Jan: SWISH. https://swish.swi-prolog.org/. Last access: September 6, 2022.

[Wi12] Wielemaker, Jan; Schrijvers, Tom; Triska, Markus; Lager, Torbjörn: SWI-Prolog. Theory and Practice of Logic Programming, 12(1-2):67–96, 2012.

[Wi19] Wielemaker, Jan; Riguzzi, Fabrizio; Kowalski, Robert A.; Lager, Torbjörn; Sadri, Fariba; Calejo, Miguel: Using SWISH to Realize Interactive Web-based Tutorials for Logic-based Languages. Theory and Practice of Logic Programming, 19(2):229–261, 2019.